A Random-Based Solitaire Game

Luca Colaiacomo*

Computer Programming project - January 2024

Abstract

In this document I will show a simulation-based solution of a solitaire game I incurred in as a kid, which I have never won in real life. Given my lifelong streak of losses,I asked myself whether a solution existed in the first place: being a deterministic combinatorial problem, no matter how uncommon it is to win, we are sure that winning is possible.

The rules of the game are the following: as in any solitaire, the aim is to sort out a randomly shuffled deck of cards,making a subdeck for each suit — Coppe, Denara, Bastoni e Spade in the Italian culture — and drawing 3 cards from the top and pulling out just the last one of these. At this point if this card is useful, i.e. is the next card in the sequence of any subdeck, it must be placed on top of that stack, otherwhise, in case of an useless card, you must draw again, until you finish your starting deck.

If you have not pulled any useful card in this round, repeating the procedure won't help, since you will end up in an infinite loop, therefore you incurred in a (not so infrequent) loss.

Otherwhise, you have to start over with the remaining unsorted deck, following the same pattern.

1. How big is the space we are playing in?

the short answer would be *huge*, but let me explain better, finding an useful representation of what actually this means in our case. To begin with, *all* the possible combinations of a deck of 40 cards are equal to:

$$40! = 40 \cdot 39 \cdot 38 \cdot ... \cdot 3 \cdot 2 \cdot 1 \approx 8.159E + 47 \tag{1}$$

at first, it is difficult to grasp what an order of 10^{47} looks like.
In order to do so, imagine to stack a sheet of paper for each combination: you would get a pile much larger than the observable universe, much more larger indeed,creating an unbalance comparable of the one between the size of a grain of sand compared to our solar system[1].

*luca.colaiacomo@studio.unibo.it - Quantitative Finance student at University of Bologna

[1]Observable universe span $\approx 8 \cdot 10^{23}$ km, a piece of paper is 0.1mm thick, a grain of sand is at most 2mm long, and the Sun-Neptune distance is $\approx 2 \cdot 10^{10}$ meters.

Figure 1: Combinations at Universal scale.

2. Coding the Simulation

Now that we have in mind the practical impossibility to reach a closed form solution, we can move to a simulation approach, namely running the game thousands of times and see what happen in the long run.
From an implementation point of view, we have to size our problem in sub-problems, that can be individually addressed in Python. The following are the issues that must be taken into account:

1. Which data type is the best suited for a deck of cards;
2. How to actually code the rule of the game, how to iterate on a deck , whose length is diminishing round after round;
3. How (and where) to put upper bounds and useful constraints.

For the first point, the most effective way to go is to think about the deck as a list of cards. Every card is different and uniquely identified by its numerical value and its suit, therefore we can represent each card as a Card Object with two arguments (value and suit): finally the deck is just a list of Card Objects.

```
class Card(): #defining the Card Object
    def __init__(self, value, suit):
        # value = 1,2,...,10 ; suit = 'Spade',...,'Bastoni'
        self.value = value
        self.suit = suit
    def __str__(self):
        return str(self.value) + ' ' + str(self.suit)

# Deck creation
num = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
symbol_list = ['Spade' , 'Coppe', 'Denara' , 'Bastoni']
deck = []
for x in num:
    for y in symbol_list:
        deck.append(Card(x, y))
```

The second point is the trickiest, and therefore needs to be furtherly broken apart. Firstly, for each suit we want a subdeck, which, in case of victory will be end up as a list from 0, starting value, to 10. Then, when we **pick** a card (more on that later) it can be a useful one or useless card. In the first case we want two things to happen: the card must be removed from the deck and the list of the corresponding suit must be updated, e.g. if we pull a 3 of *Bastoni* after having previously drwn in the game the ace and the two, we want the *bastoni-deck* to be updated from $[0, 1, 2]$ to $[0, 1, 2, 3]$.

At this point you want to update also your **pick** appropriately (now you have removed one card from the deck, therefore items positioning has shifted by -1). In case of a useless card, you simply move on with the next pull.

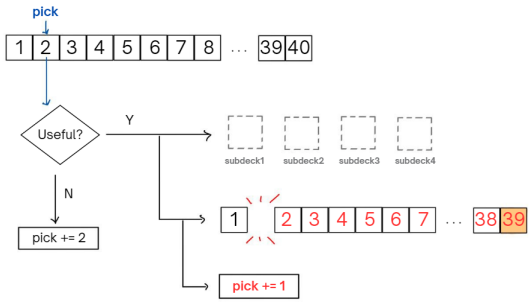


Figure 2: Pulling Algorithm.

In coding terms:

```python
# Cases in which you pull a useful card
if pick < len(deck) and deck[pick].suit == 'Spade' and int(deck[pick].value) == spade_deck[-1] + 1:
    spade_deck.append(spade_deck[-1] + 1 )
    deck.remove(deck[pick])
    pick += 2 - 1 #-1 because you have removed a card from the deck

elif pick < len(deck) and deck[pick].suit == 'Coppe' and int(deck[pick].value) == coppe_deck[-1] + 1:
    coppe_deck.append(coppe_deck[-1] + 1)
    deck.remove(deck[pick])
    pick += 2 - 1 #-1 because you have removed a card from the deck

elif pick < len(deck) and  deck[pick].suit == 'Denara' and int(deck[pick].value) == denara_deck[-1] + 1:
    denara_deck.append(denara_deck[-1] + 1)
    deck.remove(deck[pick])
    pick += 2 - 1 #-1 because you have removed a card from the deck

elif pick < len(deck) and  deck[pick].suit == 'Bastoni' and int(deck[pick].value) == bastoni_deck[-1] + 1:
    bastoni_deck.append(bastoni_deck[-1] + 1)
    deck.remove(deck[pick])
    pick += 2 - 1 #-1 because you have removed a card from the deck

# Case in which you pull a useless card
else:
    pick += 2 # here we do NOT subtract 1 because we do NOT remove any card from the deck
```

At this point, we want to repeat this process until the end of the deck is reached, and then again and again until we finish the deck. To avoid infinite loops, we must put an upper limit to the number of iterations the code must perform.
To do this, we need to think about the worst case scenario: in our case the slowest victory occurs when we pull just one useful card every time we re-explore the deck. Drawing 3 card every time from a 40 cards deck, we perform $13 + 1$ draws — you can always draw even the last card in the deck — then, from a 39 cards deck, we perform 13 draws, then with 38, 37, 36 we have 12 draws, and so on until we reach the bottom, as summarized in the table:

# Cards left	# Draws
40	14
$39 - 37$	13
$36 - 34$	12
...	...
$9 - 7$	3
$6 - 4$	2
$3 - 1$	1

At the end of the game, no matter the outcome, we would have run a number of draws which is lower or equal to the sum of the second column, namely 287.

2.1 Upper Bound

Clearly, the above problem can be generalized, obtaining a well defined closed-form solution for the upper bound. Being n the numbers of cards in the deck and m the length of the deck, and sticking to the Python notation ('//' for the integer part of a division and '%' for the remainder), we can derive d, the maximum number of draws in the game:

$$\begin{aligned} d = & \underbrace{(m\%n) \cdot (m//n + 1)}_{\substack{\text{you can pull always the last} \\ \text{card at every deck exploration}}} + n \cdot \sum_{i=1}^{m//n} i \\ & = (m\%n) \cdot (m//n + 1) \cdot + n \cdot \frac{(m//n + 1) \cdot (m//n)}{2} \\ & = (m//n + 1) \cdot (n \cdot \frac{m//n}{2} \cdot m\%n) \end{aligned} \tag{2}$$

Interestingly, we can furtherly investigate the span of this boundary:

$$d = \begin{cases} m & \text{if } n \geq m \\ 1 + 2 + ... + m = \frac{m(m+1)}{2} & \text{if } n = 1 \end{cases} \tag{3}$$

2.2 Coding the Game

Now, let's move on focusing on other technical aspects, such as the crucial end game phase, in which you will have the deck with less card than how many you want to draw, and how to re-explore a deck once reaching the bottom;
regarding both issues, we want to avoid an *out of range* error, therefore, bearing in mind that **pick** is our pointer variable, we can satisfy this conditions as follows:

```
            #in end game you will have a deck of cards smaller than how many card you draw
            if len(deck) < cards_draw:
                pick = len(deck) - 1

            if pick > len(deck) : #resetting the pointer after you reach the bottom of the deck
                pick = cards_draw - 1
```

putting all together, we can express our game in this way:

```
        while counter < upper_bound and len(deck) > 0:
            #in end game you will have a deck of cards smaller than how many card you draw
            if len(deck) < cards_draw:
                pick = len(deck) - 1

            if pick > len(deck) : #resetting the pointer after you reach the bottom of the deck
                pick = cards_draw - 1

                #Cases in which you pull a useful card
            if pick < len(deck) and deck[pick].suit == 'Spade' and int(deck[pick].value) == spade_deck[-1] + 1:
                spade_deck.append(spade_deck[-1] + 1 )
                deck.remove(deck[pick])
                pick += cards_draw - 1 #-1 because you have removed a card from the deck
            elif pick < len(deck) and deck[pick].suit == 'Coppe' and int(deck[pick].value) == coppe_deck[-1] + 1:
                coppe_deck.append(coppe_deck[-1] + 1)
                deck.remove(deck[pick])
                pick += cards_draw - 1 #-1 because you have removed a card from the deck
            elif pick < len(deck) and  deck[pick].suit == 'Denara' and int(deck[pick].value) == denara_deck[-1] + 1:
                denara_deck.append(denara_deck[-1] + 1)
                deck.remove(deck[pick])
                pick += cards_draw - 1 #-1 because you have removed a card from the deck
            elif pick < len(deck) and  deck[pick].suit == 'Bastoni' and int(deck[pick].value) == bastoni_deck[-1] + 1:
                bastoni_deck.append(bastoni_deck[-1] + 1)
                deck.remove(deck[pick])
                pick += cards_draw - 1 #-1 because you have removed a card from the deck

            #Case in which you pull a useless card
            else:
                pick += cards_draw #we do NOT subtract 1  in case of a useless card: we do NOT remove any card from the deck
            #next draw
            counter+=1
            #Winning condition
            if len(deck) == 0:
                win += 1
```

3. Running the Code

At this point, we want to check both the time it takes the code to run and whether this time is affected by the cards you draw each time, or the length of our deck (you could also try to win this game using a Poker deck, instead of an Italian one).Therefore, generalizing our code both in *len(deck)* and in *cards_draw* we can check how much does it take to run 100 trials, drawing 2, 3 or 4 cards with a deck made of a number of cards that must be a multiple of 4 — we want at least a card for each suit and every suit must have the same number of cards.

Sticking to these constraints, we get:

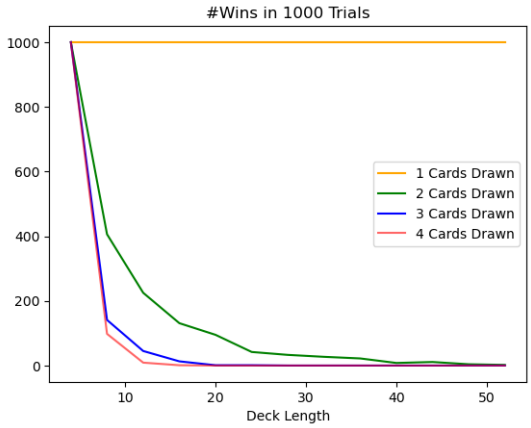
Given the low probability of winning this game — more on that in the next section — especially with 3 and 4 cards drawn, the execution time is driven mainly by the upper bound d, which, in theory,is a decreasing function of n, so the more you draw, the faster you lose: as shown in the plot, this intuitive property is more or less reflected in practice, given the 4 Cards Drawn Line laid at the bottom of the other two, apart from random spikes (at each run of the code the plot will change, but its overall behavior remain the same).

4. How Un-Likely is to Win

Lastly, and most importantly, we want to address the original question: 'How is it likely to win the game?'
As in many things in life, the answer is 'it depends', both on the length of your deck — Italian, Poker, or even a custom one — and on how many cards you draw. Regarding the first relation, we could expect that a shorter deck is easier to sort, and the less we draw the better, leading to the limit case in which we draw just one card each time — applying de facto a reverse Insertion Sort algorithm[2] — that guarantees the win.

[2]Insertion sort iterates, consuming one input element each repetition, and grows a sorted output list. At each iteration, insertion sort removes one element from the input data, finds

Let's see how these two effects interact:



Noticeably, there is an exponential decay in the number of victories, as you enlarge the deck you are trying to sort. In addition to this, drawing an additional card shifts down the plot.
Numerically, we can also compare how many times victories occur with an Italian deck (40 Cards):

Trials	2 Cards Drawn	3 Cards Drawn	4 Cards Drawn
10^2	0	0	0
10^3	13	0	0
10^4	118	0	0
10^5	1138	5	0
10^6	11249	33	0
Win Rate	$\sim 1.125 \cdot 10^-2$	$\sim 3 \cdot 10^-5$	–

Interestingly, in 1 million trials we have not managed to get even a single win with a 4 cards draw. Also, it seems that drawing an additional card leads to a ~ 1000 times lower chance of victory, so drawing 4 cards per time we should go through at least 100 million trials to get a win, more likely than a win at the lottery though.
for a complete review of the code check:
`https://github.com/cola-luke/Random-Solitaire`

the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.